Experiencias en reuso de Sistemas Multi-agente en Java

Henri Avancini
Departamento de Informática y Estadística, Universidad Nacional del Comahue
Buenos Aires 1400 - (8300) Neuquén, Argentina
E-mail: havancin@uncoma.edu.ar

Abstract

El desarrollo de Sistemas Multi-agente es un problema complejo, su resolución generalmente involucra el análisis de problemas típicos de sistemas distribuidos, con entidades (agentes) que negocian, tienen la capacidad de aprendizaje, etc. Las soluciones existentes van desde desarrollos de cero (lo que resulta muy costoso) hasta el uso de arquitecturas específicas y herramientas que asisten el desarrollo.

Se propone emplear un framework que ayuda a desarrollar Sistemas Multi-agente, permitiendo una integración gradual de los componentes y brindando una estructura de control entre ellos. Este framework se presenta utilizando como ejemplo de instanciación un agente que asiste en el manejo de una agenda personal. El diseño resultante es simple y permite la incorporación de otras funcionalidades.

PALABRAS CLAVES: Framework, Sistemas Multi-agente, Java.

1 Sistemas Multi-agente y Frameworks

Un Sistema Multi-agente (SMA) [Jennings 98] [Wooldridge 95] es un entorno que promueve las interacciones entre agentes, brindando las primitivas necesarias para esto. Un agente [Foner 93] es un programa de computadora, autónomo y dirigido por objetivos que se desenvuelve en un entorno. Autónomo significa que tiene la capacidad de realizar acciones sin la directa intervención del usuario.

Un framework en una técnica orientada a objetos de reuso [Johnson 97] . Define una estructura de clases y el control entre ellas permitiendo la reutilización de diseño y código. El mismo está implementado en un lenguaje de programación. Un framework puede ser definido a través de un conjunto de clases que establecen la estructura genérica de las aplicaciones pertenecientes a un determinado dominio. Desde esta estructura se pueden construir aplicaciones que pertenecen al dominio del framework.

La estructura de clases de cada framework posee:

(1) Métodos *totalmente implementados*, que no necesitan ser reimplementados por las subclases;
(2) Métodos *hook*, con una implementación por default que define un comportamiento determinado, estos pueden ser reimplementados por las subclases para obtener un comportamiento diferente;
(3) Métodos *template*, definen una estructura de control común a las aplicaciones que pertenecen al dominio del framework; y
(4) Métodos *abstractos*, que deben ser implementados en las subclases.

Un framework para sistemas multi-agente permite la reutilización de los componentes abstractos pertenecientes a este tipo de sistemas y sus interacciones facilitando así el desarrollo de aplicaciones. FraMaS es el nombre del framework propuesto para SMA y es el acrónimo de **Framework para Sistemas Multi-agente**. Cabe mencionar que se utiliza el lenguaje Java [Zukowski 97] para implementar el framework porque provee una arquitectura independiente de la plataforma que se desenvuelve bien en entornos distribuidos.

Este artículo está organizado de la siguiente forma. La sección 2 define la estructura de un SMA. La sección 3 es sobre el diseño de cada agente, la explicación es conducida por un ejemplo de cómo usar el framework y cómo este trabaja. La siguiente sección describe los servicios básicos de los SMA. La sección 5 describe el trabajo relacionado. Por último, en la sección 6 conclusiones y trabajo futuro.

2 Estructura de un Sistema Multi-agente

La construcción del framework fue realizada utilizando un enfoque bottom-up. Se desarrolló un SMA en Java desde el que se identificó un conjunto de componentes abstractos –y sus relaciones, a fin de ser reutilizados. La idea es motivada por la complejidad que este tipo de sistemas presenta y por la gran cantidad de componentes –y comportamientos, concretos y abstractos que pueden ser reutilizados.

En particular se diseñó un agente que ayuda al usuario en la administración de sus reuniones y tareas. Primero se construyó un sistema de agenda simple al que se le adicionaron responsabilidades. El desarrollo continuó con la identificación de los componentes generales y las relaciones entre los distintos componentes.

La estructura general propuesta para el diseño de un SMA es: (1) diseño del agente y (2) identificación de los servicios que el SMA provee para facilitar las tareas entre agentes. Por cada una ver secciones siguientes respectivamente.

La figura 1 muestra el esquema de una comunidad de agentes que pertenecen a un entorno particular, con la posibilidad de migrar de uno a otro. El entorno del SMA provee las primitivas para entrar o dejar el entorno y para conocer si otro agente se encuentra registrado.

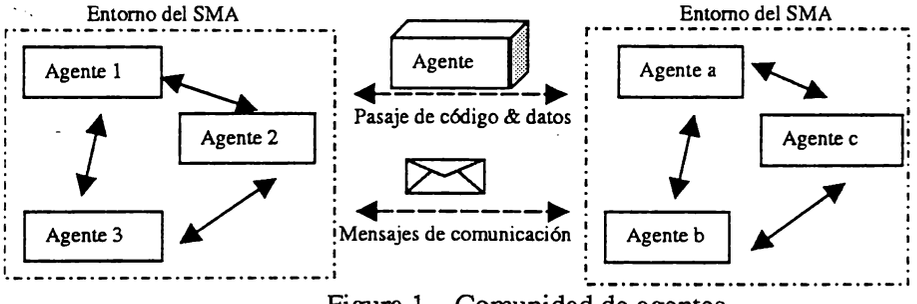

Figura 1 – Comunidad de agentes

Provee las primitivas de un servidor de nombres (almacena la dirección física actual de los agentes). Si los agentes tienen que migrar de entorno se deben implementar aquí aquellas primitivas para movilidad, en términos generales: cuando un agente quiere dejar un entorno e irse a otro guarda su estado interno –y código, solicitando al SMA el servicio para ir a otro entorno.

Cada agente se comunica directamente con otro, es decir, sin la intervención del SMA. Incluso si el agente con el que se comunica esta en otro entorno lo hace directamente. En el caso de la comunicación el servicio que brinda el SMA es para informar sobre la dirección física actual del agente.

El diseño de cada agente está basado en una estructura de clases que permite adicionar responsabilidades dinámicamente. Es decir, partir de un diseño con tareas simples y luego "cubrirlas" con nuevos comportamientos. Esta forma de agregar funcionalidad a las tareas existentes es también llamada *wrapper*.

El diseño del framework permite adicionar responsabilidades (ver en la figura 2 "comportamiento avanzado") a los objetos, sin cambiar el mensaje de llamada. Se observa en la figura el esquema de wrapper o "decoración" de la *tarea1*. Más adelante se presenta una aplicación de este diseño.

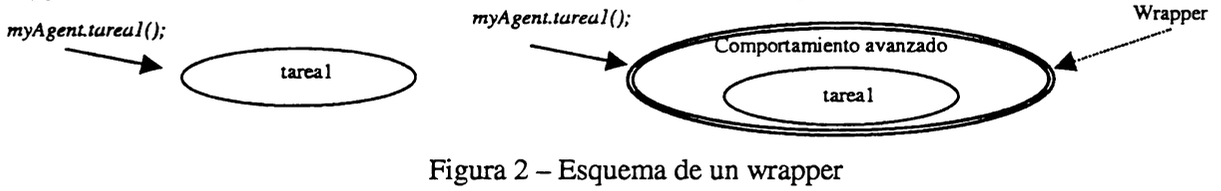

Figura 2 – Esquema de un wrapper

3 Diseño de un Agente

Cada tipo de agente es construido de la siguiente manera. Primero, se identifica, en términos generales, que será un agente en el sistema a construir. Segundo, se identifican las tareas básicas y tercero, el "comportamiento avanzado" que tendrá el agente.

Por ejemplo, en una aplicación de un asistente personal [Maes 94] para agendas, cada agente conocerá las preferencias del usuario y sugerirá de acuerdo a esas preferencias (que pueden cambiar a lo largo del tiempo).

El conjunto de acciones básicas en una agenda para la administración de reuniones son: ingresar y borrar una reunión, mostrar reuniones almacenadas, etc. (cabe mencionar que esto no es considerado un agente aún: sino que es el comportamiento básico que tendrá).

Se continua con la especificación del comportamiento avanzado. El agente agenda requiere: la capacidad de aprender las preferencias del usuario (por ejemplo usando un razonador basado en casos), negociar las reuniones con otras agendas y actuar autónomamente, es decir, que el usuario no inicie explícitamente todas sus acciones.

El framework posee una clase principal llamada *Agent*. La clase Agent utiliza un mecanismo de reflexión de Java para determinar dinámicamente las acciones que el agente puede hacer.

FraMaS posee la clase *BasicAgentAction* –hereda de Agent (ver figura 3), con el conjunto de acciones que el agente realiza y los métodos, ya implementados, para agregar y borrar acciones. Desde BasicAgentAction hereda la clase que implementa el comportamiento básico (clase Calendar en el ejemplo).

Se extiende la funcionalidad de las tareas con el objetivo de adicionar comportamientos inherentes a agentes. El pattern decorator [Gamma 94] provee la estructura de clases necesaria para permitir que estos comportamientos sean adicionados y alterados dinámicamente. Este pattern, también conocido como wrapper, es incorporado al framework. A continuación, se detalla un ejemplo que describe su funcionamiento.

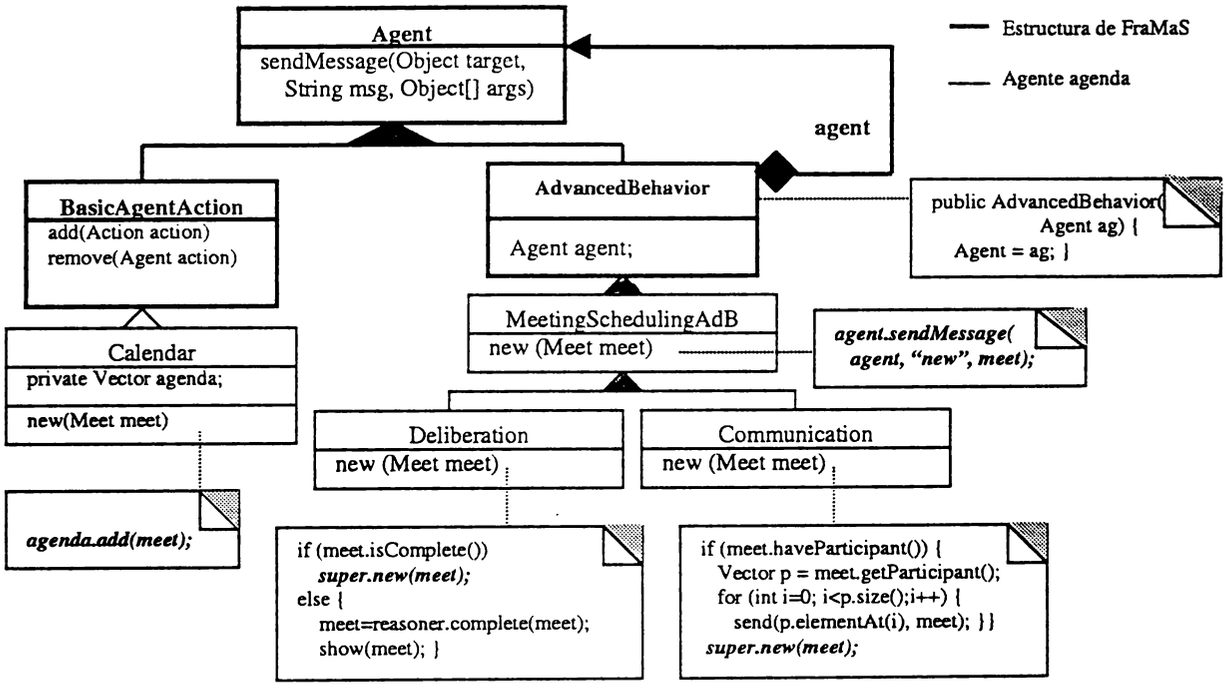

Figura 3 – Estructura de un agente en FraMaS

En la aplicación del agente agenda cada tarea tiene una fecha, hora, duración, lugar y tema; si es una reunión tendrá además una lista de participantes. Se requiere que el agente posea la capacidad de aprender las preferencias del usuario, en cuanto a: horarios usuales para cada tarea y reunión, participantes con los que trata determinados temas, lugares de reunión, etc. Como así también la capacidad de negociar las reuniones con los participantes (con la mínima intervención del usuario). Esto es expresado diciendo que los comportamientos de la agenda son "decorados" por otros comportamientos (avanzados).

Análogamente se puede seguir adicionando responsabilidades a las tareas básicas. Lo más importante es que la agregación de wrappers es trasparente, es decir, sin modificar la interface existente. Además, la estructura de control ya está implementada en el framework, por lo que sólo hay que diseñar el comportamiento particular que se desea.

Se muestra en la figura 3 un esquema del diseño del agente agenda. Se extiende desde la clase BasicAgentAction de FraMaS la clase Calendar, que implementa las acciones básicas. Luego el comportamiento básico (ingresar una cita a la agenda por ejemplo) es "decorado" por un comportamiento responsable de enviar las invitaciones a los participantes y por otro comportamiento deliberativo que completa la cita en caso de estar incompleta.

Se analiza el funcionamiento de la estructura por partes. Primero se muestra el caso de ingresar una cita dada a la agenda, y la inicialización de la misma:

```
Agent myAgent  = new Calendar();                  // No es éste un agente aún.
myAgent.sendMessage(myAgent,"new",meet);          // Ingreso de una cita a la agenda.
```

Si el usuario quiere enviar las invitaciones a los participantes en este entorno debería, por ejemplo, escribir un mail y enviarlo a cada uno de los participantes. Para adicionar este comportamiento al agente se puede usar un wrapper de comunicación sobre la agenda.

Incluso este wrapper puede negociar las reuniones con los agentes agenda de los invitados. Se implementan los métodos responsables de enviar los mensajes con las invitaciones a los participantes. También se puede usar KQML (Knowledge Query and Manipulation Language) [Finin 95], que provee un framework de comunicación con el formato de los mensajes y los protocolos para manejarlos. El primer paso está hecho, ahora para la integración hay que cambiar la inicialización:

```
Agent myAgent = new Communication(          // Wrapper de comunicación sobre ...
                        new Calendar());    // la agenda.
```

Mientras que la interface se mantiene sin modificación –el ingreso de una cita sigue siendo *myAgent.sendMessage(myAgent,"new",meet),* la estructura de control para incorporar este comportamiento esta implementada en FraMaS. Para entender cómo funciona se muestra en el ejemplo 1 la declaración de las clases *AdvancedBehavior* (de FraMaS) y *MeetingSchedulingAdB* (especialización para el agente agenda).

Ejemplo 1:

```
Public abstract class AdvancedBehavior
                                extends Agent {
  public Agent agent;
  public AdvancedBehavior(Agent ag) {
    agent = ag;
  }
}
```

```
public class MeetingSchedulingAdB
                extends AdvancedBehavior {
  public MeetingSchedulingAdB(Agent ag) {
     super(ag); }
  public void new(Meet meet){
     agent.sendMessage(agent,"new"meet); }
}
```

El mensaje *myAgent.sendMessage(myAgent,"new",meet)* es recibido por la clase Communication (ejemplo 2) y luego por la clase Calendar.

Se desea adicionar comportamiento deliberativo al diseño existente del agente a fin de aprender las preferencias del usuario y poder utilizar este conocimiento para completar las citas incompletas (liberando al usuario de la tarea de buscar espacio en su agenda, especificar todos los datos de cada cita, etc.). Para esto se emplea un razonador basado en casos: sistema para aprendizaje que utiliza la experiencia previa para resolver un nuevo problema. El entrenamiento requiere relativamente poco tiempo en relación al tiempo al hacer la consulta. El razonador utiliza los ejemplos de entrenamiento para clasificar y tratar de predecir las propiedades de un nuevo ejemplo.

El aprendizaje se realiza a través de los ejemplos de entrenamiento cargados cuando el usuario realiza una interacción con su agenda (ingresando, borrando o modificando una reunión). Se almacena la interacción como un caso nuevo. Esto se logra porque el agente esta constantemente observando las operaciones del usuario (ver *Autonomía usando Observer/Observable*).

Se implementa un método que comprueba si la cita está completa. Si no lo está usa el razonador para completarla y proponerla al usuario. Al igual que con el wrapper anterior se cambia la inicialización del agente, conservando la interface y dejando la integración al framework:

```
Agent myAgent = new Deliberation(                    // Wrapper deliberativo sobre ...
        new Communication(                           // wrapper de comunicación sobre ...
                new Calendar()));                    // la agenda.
```

El mensaje es recibido por el objeto de la clase Deliberation, luego, si la cita esta completa, lo reenvía al objeto de la clase Communication. Por último se efectúa la tarea básica que efectivamente guarda la cita en la agenda.

En el ejemplo 2 se muestra un esquema de la declaración de la clase Deliberation. Como puede observarse es el mismo esquema que en la clase Communication.

Ejemplo 2:

```
public class Communication
        extends MeetingSchedulingAdB {
  public Communication(Agent ag) {
    super(ag); }
  public void new(Meet meet) {
    if (meet.haveParticipant()) {
      Vector p = meet.getParticipant();
      for (int i=0; i< p.size(); i++) {
        msg = make_message(meet, p.elementAt(i));
        log.save("sent: "+msg+" at "+new Date());
        send(msg):
    }}
    super.new(meet);  } }
```

Comportamiento agregado

```
public class Deliberation
        extends MeetingSchedulingAdB {
  public Deliberation(Agent ag) {
      super(ag);  }
  public void new(Meet meet) {
      if (meet.isComplete())
        super.new(meet);
      lse {
        meet = reasoner.complete(meet);
        show(meet);
      }
  }
```

De la misma manera se puede seguir adicionando comportamientos al agente en forma transparente, porque no se modifica lo anterior y porque además la estructura de control para estas incorporaciones está implementada en el framework.

Autonomía usando Observer/Observable

FraMaS provee un mecanismo para detectar cambios en sus objetos, de forma que si uno de ellos cambia todos los objetos que estén subscriptos a éste recibirán una notificación. Para esto se emplea la arquitectura model-view-controler implementada en Java a través de la clase Observable y la interface Observer.

Toda clase Java que hereda de Observable puede ser observada por objetos que implementen la interface Observer. La condición es que cuando el objeto observado cambie llame al método heredado

notifyObservers(). Esto causa que se envíen mensajes a todos los métodos update() de los objetos registrados.

La clase Agent de FraMaS hereda de Observable, es decir que cualquier objeto del framework puede ser observado por otro que implemente la interface Observer. El objeto observador debe: registrarse e implementar el método llamado update(), donde efectuará las acciones.

Por ejemplo, en el sistema del agente agenda, se observan las interacciones entre el usuario y la agenda para formar la base de casos con las preferencias. Entonces, en la clase Deliberation se implementa la interface Observer, definiendo en un método llamado update() las acciones para guardar las interacciones.

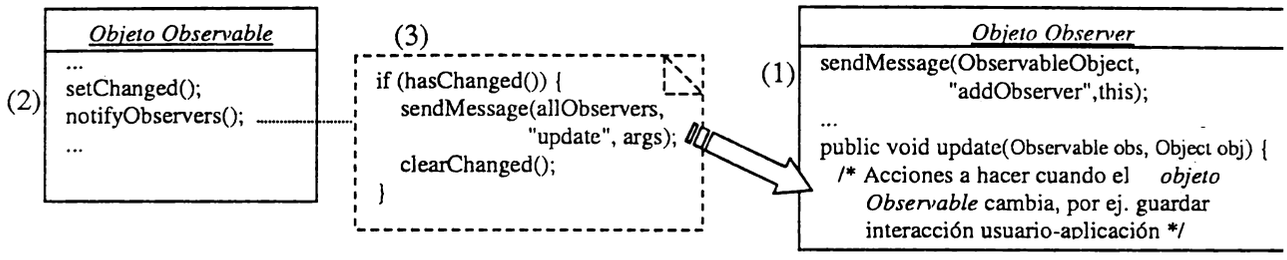

Figura 4 - Ejemplo Observer/Observable en FraMaS

Como se muestra en la figura 4, cada *objeto Observer* se registra como tal enviando un mensaje a un objeto observable (1). A continuación, cuando un objeto Observable cambia de estado lo notifica a través de los métodos setChanged() y notifyObservers() (2). Este método, verifica si el objeto ha cambiado y envía el mensaje update() a todos los objetos observer registrados (3).

Estrategias de decisión usando un thread de control independiente

Los agentes tienen una ejecución autónoma, es decir pueden efectuar tareas sin la directa intervención del usuario. Además de los wrappers y los objetos observadores, FraMaS implementa en la clase AdvancedBehavior la interface Runnable. Esto permite que una clase derivada de ella (por ejemplo MeetingSchedulingAdB) reimplemente el método run() de la interface, el cual se ejecutará independientemente.

Una estructura de control común a los agentes se puede definir a través de un método template –run(). Los agentes utilizan una estrategia de decisión, por ejemplo invocacando a un método abstracto del que se espera retorne una acción (o conjunto de acciones) que deberían ser ejecutadas en el tiempo siguiente por el agente. Los algoritmos de decisión se abstraerán en una clase abstracta que puede ser subclasificada para así definir los algoritmos concretos que respeten los protocolos en ella definidos. Un algoritmo de planning puede ser implementado como una estrategia alternativa de decisión.

Ejemplo 3:

```
public abstract class AdvancedBehavior
                    extends Agent implements Runnable {
  public Agent agent;
  public AdvancedBehavior(Agent ag) {
    agent = ag;
    if (actionThread == null) {
      actionThread = new Thread(this, "Action");
      actionThread.start(); } }
  public void run() {
    Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
    /* Selección de la prox.acción */ } }
```

El método run(), en la *clase AdvancedBehavior,* define la interface que debe usar cada agente en la selección de la próxima acción. En el ejemplo del agente agenda, el método run() es implementado en la clase MeetingSchedulingAdB, para el envío de mensajes a los participantes de las reuniones.

4 Servicios del Sistema Multi-agente

Cada agente está registrado en un sistema multi-agente. Este le provee el entorno en el que desempeña sus actividades, seguramente interactuando con otros agentes.

La clase MAS_Services de FraMaS tiene implementada la funcionalidad básica de un entorno de agentes. Esta es: registrarse y dejar el sistema, como así también preguntar si otro agente está en el entorno. Los agentes pueden interactuar con otros agentes de su entorno o de uno diferente en forma directa, es decir, sin la intervención del SMA. Para comunicarse el sistema emplea la llamada a procedimiento remoto del lenguaje Java (Remote Method Invocation).

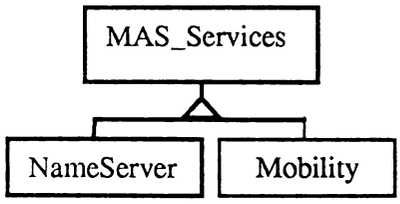

Figura 5 - Servicios del SMA

La clase MAS_Services de FraMaS es extendida para implementar un servidor de nombres y disponer de las direcciones físicas de cada agente, la clase es llamada NameServer. Los agentes se comunican con el servidor de nombres solicitando la dirección física actual de un otro agente en el sistema, para esto utilizan el nombre de usuario al que pertenece el agente, el cual no cambia.

Por otro lado la clase Mobility, especialización de la clase principal de los sistemas multi-agente (MAS_Services), será la responsable de trasladar un agente de un entorno a otro. Cuando un agente detiene su ejecución -stop(), estado "Detenido", puede solicitar al sistema que lo envíe a otro sitio. El sistema establece la comunicación con el destino quien si recibe al agente reactiva la ejecución -start(), estado "Ejecución", caso contrario se mantiene en el origen reactivando su ejecución con un código de error correspondiente.

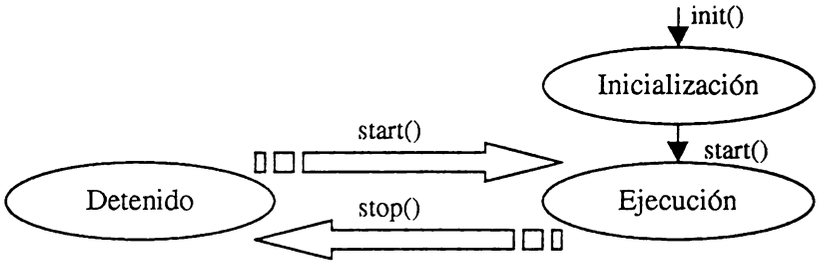

Figura 6 - Diagrama de estado de un agente

La clase principal Agent de FraMaS, tiene los métodos abstractos init(), start(), stop(); invocados para realizar la inicialización del agente, iniciar la ejecución y detenerla guardando su estado interno respectivamente. En la clase Mobility se deben implementar los métodos para transferir y recibir un agente de un entorno a otro.

5 Trabajo relacionado

Los patterns representan soluciones de diseño a problemas recurrentes en el desarrollo de software. [Gamma 94] define una serie de estos patterns. Sin embargo, los patterns son ideas de diseño de partes de un sistema. La perspectiva tomada aquí en la utilización de patterns en el framework. Usando frameworks reutilizamos diseño y código. La propuesta de [Johnson 92] de documentar los frameworks utilizando patterns facilita la futura comprensión del mismo.

Un SMA [Jennings 98] debe proveer los mecanismos suficientes para que los agentes [Foner 93] interactuen. Para la comunicación entre agentes [Finin 95] se puede utilizar el lenguaje KQML, que define la formato de los mensajes y los protocolos de comunicación.

El agente agenda que se describió con FraMaS no es un agente que se desplace de un host a otro, aunque otro podría hacerlo, los aspectos de movilidad son tratados en [Karnik 98]. En general el esfuerzo de desarrollo en el área es conducido hacia facilitar la construcción de SMA para evitar su desarrollo desde cero.

6 Conclusión y trabajo futuro

El uso de este framework ayuda en la construcción de SMA evitando tener que desarrollarlos desde cero. Los frameworks permiten reutilizar diseño y código, con lo que se reducen los costos de producción de sistemas (pertenecientes al dominio del framework). El código puede ser empleado para una rápida implementación de la aplicación a desarrollar utilizando los métodos ya definidos y luego, redefiniendo algunos es posible personalizar la aplicación según los requerimientos particulares.

FraMaS provee la estructura de clases y control necesarios para adicionar responsabilidades dinámicamente (pattern Decorator), puede disponer de objetos "observadores" sobre cualquier otro objeto derivado del framework, y un thread para la selección de la próxima acción.

El uso de un framework para SMA es útil para establecer una definición común de qué es un SMA. Como tiene definida una estructura de clases y cómo sus instancias interactuan, define un comportamiento que cumplen los SMA. Sin embargo, los frameworks tienen la desventaja de ser de difícil comprensión al comenzar a utilizarlos. Para solventar esto se emplean patterns (que solucionan problemas de diseño particulares) para su documentación y facilitar así su utilización.

Permite la identificación de componentes comunes a los SMA y su reuso en otros sistemas. El uso actual de FraMaS ha mostrado si viabilidad como ayuda en la construcción de SMA. El trabajo futuro es continuar validando estas abstracciones con otras instanciaciones.

7 Referencias

[Atherton 98] Atherton, R. Moving Java to the Factory. IEEE Spectrum. December 1998.

[Eckel 98] Eckel, B. Thinking in Java. Prentice Hall, 1998.

[Ferguson 94] Ferguson, I. Integrated Control and Coordinated Behaviour: a Case for Agent Models. Intelligent Agents. Berlin: Springer-Verlag, 1994.

[Finin 95] Finin, T. et. al. KQML as an agent communication language. Software Agent. Ed. Jeff Bradshaw, MIT Press, Cambridge, 1995.

[Foner 93] Foner, L. What's an agent, anyway? A Sociological Case Study. Massachusetts: MIT Media Laboratory, May 1993.

[Gamma 94] Gamma, E. et. al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[Jennings 98] Jennings, N. et. al. A Roadmap of Agent Reasearch and Development. Autonomous Agents and Multi-Agent Systems. Vol. 1, No. 1, 1998.

[Johnson 92] Johonson, R. Documenting Frameworks using Patterns. In Proceeding of the O.O.P.S.L.A., 1992.

[Johnson 97] Johnson, R. Components, Frameworks, Patterns. In: Symposium on Software Reusability, Proceedings...,p.10-17, 1997.

[Karnik 98] Karnik, N. and Tripathi, A. Design Issues in Mobile-Agent Programing Systems. IEEE Concurrency, July/September 1998.

[Maes 94] MAES, P. Agentes that can reduce work and Information overload. Communications of the ACM. July 1994 – Volume 37, Number 7.

[Meyer 96] Meyer, B. Object Oriented Software Construction. 2/E. Prentice Hall PTR, 1996.

[O'Hare 96] O'Hare, G. and Jennings, N. (Eds.) Foundations of Distributed Artificial Intelligence. ISBN 0-471-00675-0, 1996.

[Shaw 96] Shaw, M.; Garlan, D. Software Architecture. Perspectives on an Emerging Discipline. New Jersey: Prentice Hall, 1996.

[Wooldridge 95] Wooldridge, M.; Jennings N. Agent Theories, Architectures, and Languages: A Survey. Intelligent Agents. Berlin: Springer-Verlag, 1995.

[Zukowski 97] Zukowski, J. Mastering Java 1.2. Ed. Sibex, 1997.